Representing and analyzing change in a formal modular specification

V. Wiels and S. Easterbrook
{wiels,steve}@atlantis.ivv.nasa.gov

NASA/WVU Software Research Lab
100 University Drive, Fairmont WV 26554, USA
fax: 1 304 367 8211

Abstract. We are interested in the formal modeling of change requests for large systems. In large systems development, changing requirements are a fact of life. If formal methods are to be successfully applied in such a context, they must be able to handle evolving specifications, to reason about proposed changes, thus helping to validate the change requests. In this paper we propose a method to represent change requests and analyze their impact on the existing specification. We use a category theory based framework to specify the system in a modular way and to analyze changes. The approach is applied to a software change request for the Space Shuttle.

1 Introduction

We are interested in the formal modeling of change requests for large systems. Very little attention has been paid in the formal methods community to the management of change during systems development. Hence most case studies of formal methods have concentrated on the use of a formal specification as a baseline from which designs and implementations can be verified. In large systems development, changing requirements are a fact of life. If formal methods are to be successfully applied in such a context, they must be able to handle evolving specifications, to reason about proposed changes, thus helping to validate the change requests.

Structure is essential for managing large-scale specifications [4]. A well-chosen structure greatly facilitates understanding and modification of a specification [8]. Category theory has been proposed as framework for providing this structure, and has been successfully used to provide composition primitives in both algebraic [14, 3] and temporal logic [6, 5, 11] specification languages. Category theory is ideal for this purpose, as it provides a rich body of theory for reasoning about objects and relations between them (in this case, specifications and their interconnections). Also, it is sufficiently abstract that it can be applied to a wide range of different specification languages. Finally, it lends itself well to automation, so that, for example, the composition of two specifications can be derived automatically, provided that the category of specifications obeys certain properties (e.g. co-completeness).

Work on category theory for software specification has typically adopted a "correct by construction" approach: components are specified, proved correct and then composed together in such a way to preserve all their properties. This kind of framework is too constraining and needs to be adapted and relaxed in order to be able to handle changes. In [13], we propose some extensions to deal with evolving specifications and especially the management of properties of such specifications. But we also need to address the problem of actually representing changes.

In this paper, we propose an approach to model changes in a modular formal specification and to analyze their impact on the existing specification. We use a new categorical framework proposed by Lopes and Fiadeiro in [7]. This framework extends the classical notions of specification and specification morphism by differentiating two kinds of properties for a component: axioms that are true in any environment and co-axioms that depend on the collaboration of other components of the system. This framework is very general and thus allows us to consider different interconnection models. We are using this framework to specify a system in a modular way and to represent changes and their relationships with the existing specification.

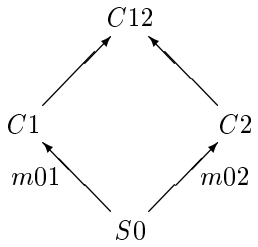
We apply our approach to a software change request for the Space Shuttle. Space Shuttle change requests offer an excellent testbed to explore the issues raised by evolving specifications. The shuttle flight software is an extremely well documented system, where a complete set of revised specifications is available for each version of the software, and the lifecycle of each change request is fully documented.

The paper is organized as follows. Section 2 gives an overview of the approach. Section 3 presents the Space Shuttle software and change requests. Section 4 gives a user-oriented overview of the categorical framework. In section 5, the change request is specified and analyzed with this categorical framework. Section 6 gives some conclusion and sketches avenues of future work.

2 Overview of the approach

As pointed out in [13, 8], dealing with changes is much easier with a structured specification. A well chosen structure helps to circumscribe the parts of the specification that must be modified more easily and to evaluate the consequences of the changes.

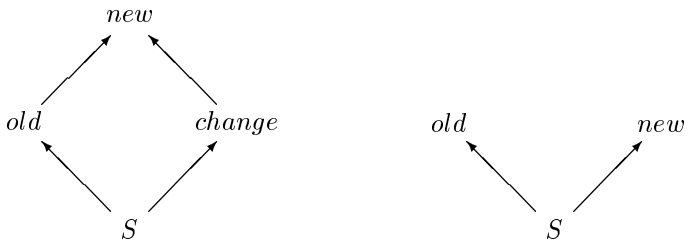
Category theory is very well suited to specify systems in a modular way. In a typical category theory based framework, the specification of a system is a diagram with a specification for each component and specifications and morphisms representing the relationships between the components. The basic diagram for two components is the following:



C1 and C2 are two components, S0, m01 and m02 specify the relationships between them, and by applying a categorical operation called pushout to this diagram, we get C12 which represents the composition of C1 and C2.

In classical category theory based frameworks, morphisms must preserve all properties from the source specification to the target one. This is a strong assumption and Lopes and Fiadeiro argue in [7] that it needs to be relaxed to better specify reactive systems. We also think that a relaxed framework is necessary to represent and analyze evolving specifications. The framework proposed by Lopes and Fiadeiro differentiate two kinds of properties for a component: axioms that are true in any environment and co-axioms that depend on the collaboration of other components of the system. The framework allows to consider different interconnection models and is very general.

This framework is flexible enough to deal with changes. In [7], an example is given where a specification is modified and the new specification is computed from the old specification and the change as shown on the left diagram below:



This is an ideal case, the change is then just considered as a new component added to the system. However, this solution is not possible in all cases. The proposed change is, in some cases, contradictory with the existing specification. The change request presented in this paper is an example of such a change.

We claim that the categorical framework is still useful in those cases. We propose to adopt an alternative approach: we consider the diagram on the right in the figure above. When we compute the pushout of this diagram in the relaxed framework (with a suitable interconnection model), we obtain the part of the old specification that is compatible with the change. Moreover, by computing the difference between the properties of the new specification and the properties in the pushout, we get the modifications that are not consistent with the old behavior.

In the following, we present the approach in more details and apply it to a software change request for the Space Shuttle.

3 SPACE SHUTTLE SOFTWARE AND CHANGE REQUESTS

3.1 Space Shuttle software

As an operational vehicle, the Space Shuttle regularly needs updates to its flight software to support new capabilities (such as docking with the space station), replace obsolete technology (such as the move to GPS for navigation), or to correct anomalies. Software updates are known as Operational Increments (OIs), and are typically completed approximately every twelve to eighteen months. An OI will implement any number of change request (CRs). A change request typically consists of a selection of pages from current Computer Program Design Specification (CPDS) and Functional Subsystem Software Requirements (FSSR) specifications, with handwritten annotations showing new and changed requirements.

Each change request is reviewed by a number of requirements analysts, along with members of the Independent Verification and Validation team, culminating in a formal requirements inspection. Following this inspection, the change request may be rejected, revised for re-inspection, or forwarded to the review board for inclusion in the current OI.

3.2 ECAL Change Request

The case study concentrated on change request #90724, the East Coast Abort Landing (ECAL) automation CR. The change request covers changes needed to automate the entry guidance procedures for an emergency landing at sites on the East Coast or Bermuda, following a loss of thrust during launch, such that orbit cannot be attained. The rationale for automating these procedures is that it will reduce the costs of crew training, and increase the probability of successful landing. The core functionality of the change request covers the management of shuttle's energy during descent and the guidance needed to align it with the selected runway.

Our approach is to model the old requirements in the FSSR first, and then update this model to reflect the changes listed in the CR. The ECAL change request affects seven specifications, but we chose to concentrate only on one FSSR, namely STS-83-0001-27 "Entry Through Landing Guidance". The changes all referred to the phase G4.204 of the entry requirements, namely "Return-to-launch site (RTLS) Terminal Area Management (TAEM) guidance". This represented approximately two-thirds of the change request and was selected because it contained the core of the new functionality.

3.3 RTLS guidance

The FSSR requirements for RTLS guidance are structured in functions. The main function GREXEC is executed at each time cycle and calls 13 other functions depending on the value of *iphase* (a variable representing the current guidance phase). The different functions exchange data between themselves and with

the environment. Each function is described by general requirements in natural language, detailed requirements in pseudo-code and tables giving the inputs, outputs and constants of the function. Further details of these requirements will be presented in section 5.

4 CATEGORICAL FRAMEWORK

Traditional categorical frameworks use a strong notion of morphism: a specification morphism has to preserve all properties, meaning that when a component is embedded in a system, it still has all the properties it had when considered alone. This is a strong assumption and it is not always adapted to the specification of reactive systems. In [7], Lopes and Fiadeiro propose new notions of specification and specification morphism. The behavior of a component is described by two types of sentences: axioms and co-axioms. Axioms express properties that the component always has, co-axioms give properties that are true when the component is considered alone but that can depend on the collaboration of other components when it is embedded in a system. Morphisms preserve axioms but only reflect co-axioms (i.e. resulting co-axioms are the properties on which all the components agree).

We will use this framework to specify the system, but also to represent and analyze change request on the system. In this section, we give a brief user-oriented presentation of the framework. For more details about the theoretical aspects, we refer the reader to [7]. It must also be noted that we use a first order version of the framework and that we use a slightly different notion of morphism: the action part of their morphism is reversed, which can be useful for certain type of interactions but makes the logic more complicated; we keep the action mapping in the traditional direction, because it is sufficient for the system considered here. But this shows that the new framework can be adjusted to the kind of system considered.

4.1 Specifications

A component of a system is described by a specification. A specification is a triple $< \Sigma, \Phi, \Psi >$.

$\Sigma =< DT, At, Ac >$ is a signature giving the vocabulary of the specification. DT gives the data types used in the specification with their algebraic specification. At is a finite set of attributes, attributes are local to a component. Ac is a finite set of actions, to these actions is added the action $\perp$ representing idle steps of the component, that is, the steps performed by the environment. During these steps, the attributes cannot change values (encapsulation).

Φ is a set of axioms such that $\Phi \subseteq PROP(\Sigma)$. $PROP(\Sigma)$ is the set of properties ϕ for Σ: $\phi ::= \varphi \mid (init \rightarrow \varphi) \mid (\varphi \rightarrow [acs]\varphi')$, for $acs \subseteq Ac$, $\varphi, \varphi' \in STAT(\Sigma)$,
where $STAT(\Sigma)$ is the set of state propositions: $\varphi ::= (t1 = t2) \mid (\neg\varphi) \mid (\varphi \rightarrow \varphi')$

where $t1$ and $t2$ are terms of same type.
$[acs]\varphi$ means that for each action ac in acs, φ is true after the execution of ac.

Properties capture invariants, initialisation conditions, effects of actions and restriction to the occurrence of actions. More generally, properties concern local features that will be preserved when the component is embedded in a system.

Ψ is a set of co-axioms such that $\Psi \subseteq CO{-}PROP(\Sigma)$. $CO{-}PROP(\Sigma)$ is the set of co-properties: $\psi ::= \varphi \mid (\varphi \rightarrow < ac > true)$ for $ac \in Ac$ and $\varphi \in STAT(\Sigma)$. $< ac > true$ means that the action ac can occur.

Co-properties capture the ability of actions to occur in certain states (readiness). More generally, co-properties are not preserved when the component is embedded in a system, they are only "reflected". For example, for readiness properties, a system is ready to execute an action only if all the components involved in the execution of that action are ready to execute it (actions are global and can be shared between several components).

4.2 Specification morphisms

A signature morphism σ from $\Sigma =< DT, At, Ac >$ to $\Sigma' =< DT', At', Ac' >$ is a triple $< \sigma_{DT}, \sigma_{At}, \sigma_{Ac} >$ where $\sigma_{DT} : DT \rightarrow DT'$ is a classical algebraic signature morphism (see [14, 3]), $\sigma_{At} : At \rightarrow At'$ is a mapping and $\sigma_{Ac} : Ac \rightarrow Ac'$ is a mapping. A signature morphism induces a translation on properties and co-properties, denoted by $\underline{\sigma}$ as follows:

$\underline{\sigma}(\phi) ::= (\sigma(t1) = \sigma(t2)) \mid (\neg\underline{\sigma}(\varphi)) \mid \underline{\sigma}(\varphi) \rightarrow \underline{\sigma}(\varphi') \mid init \rightarrow \underline{\sigma}(\varphi)$
$\mid \underline{\sigma}(\varphi) \rightarrow [\sigma_{Ac}(acs)]\underline{\sigma}(\varphi) \mid \underline{\sigma}(\varphi) \rightarrow < \sigma_{Ac}(ac) > true$

A specification morphism $\sigma : S =< \Sigma, \Phi, \Psi > \rightarrow S' =< \Sigma', \Phi', \Psi' >$ is a signature morphism such that:

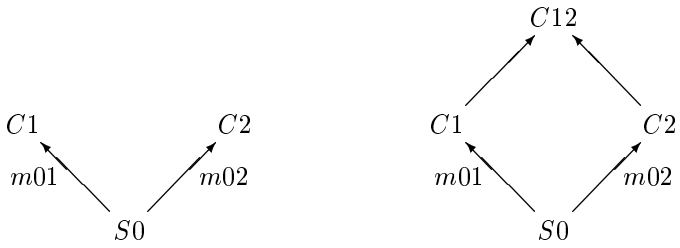
1. $\Phi' \models_{\Sigma'} \underline{\sigma}(\Phi), \underline{\sigma}(loc(\Sigma))$
2. $\Phi'^{\bullet} \cap STAT(\Sigma'), \underline{\sigma}(\Psi), \underline{\sigma}(env(\Sigma)) \models_{\Sigma'} \Psi'$

where $\Phi'^{\bullet} = \{\phi : \Phi' \models_{\Sigma'} \phi\}$; $loc(\Sigma)$ is the locality property (expressing that the attributes of a specification cannot change value during idle steps); and $env(\Sigma)$ expresses the fact that the environment can always progress (idle steps are always possible). $\underline{\sigma}(loc(\Sigma))$ means that the actions added in the target specification cannot modify the attributes coming from the source specification. $\underline{\sigma}(env(\Sigma))$ is $\forall ac \in Ac' - \sigma(Ac), < ac > true$.

A specification morphism thus preserves all the axioms of the source specification and also preserves the locality of the source specification. The rule is different for the co-axioms: a morphism only reflects them, which means that the co-axioms of the target specification are the co-properties on which all the components agree.

4.3 Interconnecting Specifications

Each component of a system is described by a specification. Then the specifications are interconnected in the following way:



C1 and C2 are two components of the system. S0 contains the elements shared by the two components, the morphisms m01 and m02 make the necessary associations (and allow renaming).

The specification of the system C12 is given by the pushout (for 2 components) or colimit (for n components) of the resulting diagram. C12 is as follows:

- the signature is the union of the two signatures of C1 and C2, but taking into account that the intersection is defined by S0;
- the axioms are the union of axioms from C1 and C2;
- the co-axioms are the co-properties that can be proved from co-axioms of C1 and from coaxioms of C2.

We can illustrate this on a very simple example:

	$C1$	$C2$
$Data:$	$Bool$	$Bool$
$Attr:$	$at1, at2 : Bool$	$at3, at4 : Bool$
$Act:$	$ac1, ac2$	$ac3, ac4$
$Ax:$	$init \rightarrow \neg at2 \wedge at1$	$init \rightarrow \neg at3 \wedge \neg at4$
	$[ac1]at2$	$[ac3]at3$
	$\neg at2 \rightarrow [ac2]false$	
$Coax:$	$at2 \rightarrow < ac2 > true$	$\neg at4 \rightarrow < ac4 > true$
	$\neg at2 \rightarrow < ac1 > true$	$< ac3 > true$

The axioms give initialization conditions, post-conditions of actions and cases where the action cannot occur. Co-axioms give the readiness conditions for the actions (for example $ac2$ can occur only when $at2$ is true while $ac3$ can occur at any time).

S0 contains only *bool* and one action $ac24$ that is associated to $ac2$ by $m01$ and to $ac4$ by $m02$. The pushout is then:

$$
\begin{array}{ll}
 & C12 \\
Data: & Bool \\
Attr: & at1, at2, at3, at4 : Bool \\
Act: & ac1, ac3, ac24 \\
Ax: & init \rightarrow \neg at2 \wedge at1 \wedge \neg at3 \wedge \neg at4 \\
 & [ac1]at2 \\
 & [ac3]at3 \\
 & \neg at2 \rightarrow [ac24]false \\
Coax: & at2 \wedge \neg at4 \rightarrow < ac24 > true \\
 & \neg at2 \rightarrow < ac1 > true \\
 & < ac3 > true
\end{array}
$$

The two components are synchronised on the execution of actions $ac2$ and $ac4$ (which become one action $ac24$). The action $ac24$ represents the simultaneous execution of $ac2$ and $ac4$. All axioms are preserved. Co-axioms are reflected: for example for the action $ac24$, the readiness condition is the one that can be proved from the translation of both readiness conditions for $ac2$ and $ac4$.

Remark: it is important to notice that names are local to each component and that two components can be associated only by means of morphisms, which means that if there are two attributes with the same name a in C1 and C2 but they are not identified by S0, m01 and m02, there will be two different attributes in C12: C1.a and C2.a.

4.4 Dealing with shared attributes

The decomposition into axioms and co-axioms depends on the interconnection model that is chosen. In what we presented before ([7]), attributes are local and actions can be shared. Consequently, axioms concern local attributes (invariants, effect of the action on attributes) and co-axioms deals with readiness conditions of actions. The attributes cannot be changed by the environment while if an action is shared, the resulting readiness condition must take into account the readiness condition of all the components involved in this action.

However, the framework is general and other models of interconnection can be considered provided that the decomposition into axioms and co-axioms is changed accordingly. In the following, we will use the interconnection model presented before, but also a different interconnection model that allows us to have shared attributes. In that case, invariants or effects of actions on attributes are co-axioms.

5 SPECIFICATION OF THE SHUTTLE CHANGE REQUEST

In this section, we first explain how to model the FSSR requirements in the categorical framework, then we propose an approach to manage the change request.

5.1 Specification of one function

We take as a typical example one of the thirteen functions of the system. The RTLS angle of attack command function (GRALPC) computes the angle of attack for the alpha recovery phase (iphase=6) and the alpha transition phase (iphase=4). It also computes an incremental normal acceleration value (dgrnz). We give below an extract of the detailed requirements for this function:

```
If IPHASE = 6, the constant alpha recovery angle-of-attack command,
ALPCMD, as well as the altitude rate dependent incremental NZ
command, DGRNZ, for the load relief phase (IPHASE = 5) are computed
as shown in Equation Set 1.
  1.1 If CONT=OFF then ALPCMD=ALPREC
  1.2 If CONT=ON then ALPCMD=MIDVAL(ALPRECS MACH+ALPRECI,ALPRECU,ALPRECL)
  1.3 If HDOT<HDMAX, then HDMAX=HDOT
Otherwise (HDOT>=HDMAX) execute Equation Set 1.3 for intact aborts
(CONT=OFF) or Equation Set 1.4 for contingency abort (CONT=ON)
     1.3.1 DGRNZ=MIDVAL((HDNOM-HDMAX)DHDNZ, DHDLL, DHDUL)
     1.3.2 DGRNZT=GRNZC1 + DGRNZ + 1.0
     1.4.1 DGRNZT=MIDVAL(DNZB - HDMAX DHDNZ, DNZMIN, DNZMAX)
     1.4.2 SMNZ1 = ZDT1 DGRNZT
     1.4.3 DGRNZ=DGRNZT - GRNZC1 - 1.0
     1.4.4 NZSW=GRNZC1-SMNZ1-SMNZ2+1.0
```

We do not give the rest of the requirements, they define ALPCMD and DGRNZ when IPHASE is different from 6. The tables tell us that IPHASE, CONT, HDOT, SMNZ2 and MACH are inputs; ALPCMD, DGRNZ, DGRNZT, NZSW and SMNZ1 are outputs; and all the others are constants.

The specification of GRALPC is as follows:

$$
\begin{array}{ll}
Data: & Bool, Int, Float, Flag \\
 & hdmax, hdnom, dhdnz, dhdll, dhdul, alprec, alprecs : Float \\
 & dnzmin, dnzmax, alpreci, alprecl, alprecu : Float \\
Attr: & gralpc, ok : Bool, cont : Flag, iphase : Int \\
 & alpcmd, dgrnz, hdot, mach, smnz2, nzsw, smnz1, dgrnzt : Float \\
Act: & begin, end \\
 & imp\_cont(Bool), imp\_hdot(Float,) imp\_iphase(Int), imp\_mach(Float), \\
 & imp\_smnz2(Float), exp\_smnz1(Float), exp\_alpcmd(Float), \\
 & exp\_dgrnz(Float), exp\_nzsw(Float), exp\_dgrnzt(Float)
\end{array}
$$

The data part gives the data types and the constants used by the component (we have omitted the specification of each data type for sake of conciseness). The attributes part contains all the variables found in the requirements plus two booleans (*gralpc* and *ok*) that will be used to specify the sequencing between the different functions of the system: *gralpc* is true when the function is being executed, *ok* is true when the function has finished its execution. The action part contains an import action for each input variable, and export action for

each output variable and two actions *begin* and *end* representing the beginning and end of the function.

$$
\begin{array}{l}
Ax : init \rightarrow \neg gralpc \ ; \ [begin]gralpc \wedge \neg ok \ ; \\
\quad \neg gralpc \rightarrow [imp\_cont(x), imp\_hdot(y), imp\_smnz2(y), \\
\quad\quad imp\_iphase(z), imp\_mach(y)]false \ ; \\
\quad [imp\_cont(x)]cont = x \ ; \ [imp\_hdot(y)]hdot = y \ ; \\
\quad [imp\_smnz2(y)]smnz2 = y \ ; \ [imp\_iphase(z)]iphase = z \ ; \\
\quad [imp\_mach(y)]mach = y \ ; \\
\quad cont = on \wedge iphase = 6 \rightarrow \\
\quad\quad alpcmd = midval(alprecs * mach + alpreci, alprecu, alprecl) \ ; \\
\quad cont = off \wedge iphase = 6 \rightarrow alpcmd = alprec \ ; \\
\quad cont = off \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow \\
\quad\quad dgrnz = midval((hdnom - hdmax)dhdnz, dhdll, dhdul) \\
\quad\quad \wedge dgrnzt = grnzc1 + dgrnz + 1.0 \ ; \\
\quad cont = on \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow \\
\quad\quad dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax) \\
\quad\quad \wedge smnz1 = zdt1 * dgrnzt \wedge dgrnz = dgrnzt - grnzc1 - 1.0 \\
\quad\quad \wedge nzsw = grnzc - smnz1 - smnz2 + 1.0 \ ; \\
\quad ... \\
\quad \neg ok \rightarrow [end]false \ ; \ [end]\neg gralpc
\end{array}
$$

The axioms part contains three different kinds of properties:

- axioms dealing with the beginning and end of the function: no action can happen if *gralpc* is false (i.e. if the function is not being executed), the action *end* is only possible when everything in the function has been done (i.e. *ok* is true);
- axioms concerning the import actions: the value of the input variables are stored in local attributes;
- axioms specifying the definition of the output variables from the input variables.

$$
\begin{array}{l}
Coax : \neg gralpc \rightarrow < begin > true \\
\quad\quad gralpc \rightarrow < imp\_cont(x) > true \\
\quad\quad gralpc \rightarrow < imp\_hdot(y) > true \\
\quad\quad gralpc \rightarrow < imp\_smnz2(y) > true \\
\quad\quad gralpc \rightarrow < imp\_iphase(z) > true \\
\quad\quad gralpc \rightarrow < imp\_mach(y) > true \\
\quad\quad alpcmd = y \rightarrow < exp\_alpcmd(y) > true \\
\quad\quad dgrnz = y \rightarrow < exp\_dgrnz(y) > true \\
\quad\quad dgrnzt = y \rightarrow < exp\_dgrnzt(y) > true \\
\quad\quad nzsw = y \rightarrow < exp\_nzsw(y) > true \\
\quad\quad smnz1 = y \rightarrow < exp\_smnz1(y) > true \\
\quad\quad ok \rightarrow < end > true
\end{array}
$$

The co-axioms give the readiness conditions of the actions. The *begin* action can occur if the function is not already being executed, the import actions can

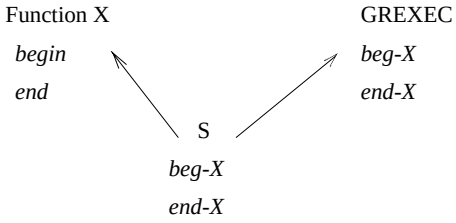
occur if the function is being executed, the export actions can export a value if the variable has this value and the *end* action can occur if everything in the function has been executed.

5.2 Specification of the system

GRALPC has several interactions with the other components of the system: it imports data from and exports data to other functions; and it synchronises with GREXEC for sequencing.

For example, GRALPC exports *dgrnz* to two other functions called GRTRN and TGNZC. Two subspecifications are thus defined between GRALPC and GRTRN on one hand and GRALPC and TGNZC on the other hand. Each specification contains an action $com\_dgrnz(Float)$ that is associated to *exp_dgrnz* in GRALPC and to the corresponding *imp_dgrnz* in GRTRN and TGNZC. This will induce two actions in the resulting system representing the communication of the value of *dgrnz* from GRALPC to GRTRN and from GRALPC to TGNZC.

The sequencing of functions is handled in the following way: the GREXEC specification has *begin_X* and *end_X* actions for each function X and describes how the functions are sequenced. The specification of each function X has corresponding *begin* and *end* actions (see for example GRALPC). The actions are synchronised as shown below:



Finally, all the components and their interactions (subspecifications and morphisms) form a diagram, the colimit of this diagram can be computed: it is a specification describing the whole system.

5.3 CR for GRALPC

In the requirements, two cases are taken into account: intact aborts (CONT = OFF) and contingency aborts (CONT = ON). In the CR, we now have to consider three cases: intact aborts (CONT = OFF), contingency aborts (CONT = ON and ECAL = OFF) and ECAL aborts (CONT = ON and ECAL = ON) where ECAL is a new input of the system.

For example, in GRALPC, the definition of *dgrnzt* is different for ECAL aborts. The change request for the part of the requirements given before is as follows:

```
Renumber 1.4.2, 1.4.3 and 1.4.4 respectively 1.4.4, 1.4.5 and 1.4.6
and insert the following:
    1.4.2 If (ECAL=ON and ABS(DPSAC)>DPSAC1 and DGRNZT<DNZMX1)
```

```
        then ITGTNZ=ON
  1.4.3 If ITGTNZ=ON then DGRNZT=DGRNZT+DNZ1
```

where ECAL and DPSAC are new inputs, ITGTNZ is a new output and DPSAC1, DNZMX1 and DNZ1 are new constants of GRALPC.

5.4 Management of the CR

Now we have specified the system in a modular way, we want to represent the CR and to get some information about its consequences with respect to the current version of the system. We first handle the CR at the component level, then consider the system level.

For GRALPC, the change request is contradictory with the existing requirements. In the FSSR requirements, if *cont* is *on* and *iphase* is equal to 6 and *hdot* is greater or equal to *hdmax*, then *dgrnzt* is always equal to $midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax)$. In the new version of the GRALPC function, this is only true if *itgtnz* is off. There is thus no way we could define a specification which, composed with the old GRALPC specification, would describe the new requirements for GRALPC.

But we can still use the categorical framework to get some information about the consequences of the change on the existing component, about the potential inconsistencies between the CR and the existing specification. We are going to compute the pushout of the two versions of the component, but to do so, we need to consider what is shared and thus what is the decomposition in axioms and co-axioms.

If we consider the two versions of a component in the case of the shuttle CR, we see that they have a lot of elements in common and we notice that they share attributes. So we have to consider the interaction in a different way and consequently change the decomposition in axioms and co-axioms. Some attributes are not local any more, they can be shared and so the axioms concerning these attributes become co-axioms.

And with this configuration, when we compute the pushout of the two versions of the component, the result gives us the part of the old version that is compatible with the change.

We can see in more details how this works on the example of the GRALPC function. The specification of the new function is as follows:

$Data:$ $old\ data \oplus dpsac1, dnz1, dnzmx1 : Float$
$Attr:$ $old\ attr \oplus dpsac : Float, ecal, itgtnz : Flag$
$Act:$ $old\ act \oplus imp\_ecal(Flag), imp\_dpsac(Float), exp\_itgtnz(Flag)$
$Ax:$ $old\ ax$
$\ominus$ $cont = on \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow$
$dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax)$
$\oplus$ $cont = on \wedge iphase = 6 \wedge hdot >= hdmax \wedge itgtnz = off$
$\rightarrow dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax)$
$cont = on \wedge iphase = 6 \wedge hdot >= hdmax \wedge itgtnz = on$
$\rightarrow dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax) + dnz1$
$cont = on \wedge iphase = 6 \wedge hdot >= hdmax$
$\wedge\ ecal = on \wedge abs(dpsac) > dpsac1$
$\wedge\ midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax) < dnzmx1$
$\rightarrow itgtnz = on$
$Coax:$ $old\ coax$
$\oplus$ $gralpc \rightarrow < imp\_ecal(x) > true$
$gralpc \rightarrow < imp\_dpsac(y) > true$
$itgtnz = x \rightarrow < exp\_itgtnz(x) > true$

We want to compute the pushout of the two versions of GRALPC in order to get some information about their compatibility. To do so, we have to define the subspecification S0. S0 contains the elements common to the two specifications, which is in fact the signature of the old GRALPC specification. But if attributes are shared as well as actions, the decomposition in axioms and co-axioms has to be redefined, and we end up in fact with only co-axioms on both side. Hence, when we compute the pushout, we get the coaxioms the two specifications agree on. For GRALPC, we can look at the formulas defining $dgrnzt$.

In the old specification, $dgrnzt$ is defined as follows:
$cont = off \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow dgrnzt = grnzc1 + dgrnz + 1.0$
$cont = on \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow$
$dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax)$

In the new specification, we have:
$cont = off \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow dgrnzt = grnzc1 + dgrnz + 1.0$
$cont = on \wedge iphase = 6 \wedge hdot >= hdmax \wedge itgtnz = off$
$\rightarrow dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax)$
$cont = on \wedge iphase = 6 \wedge hdot >= hdmax \wedge itgtnz = on$
$\rightarrow dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax) + dnz1$

In the resulting pushout, we get:
$cont = off \wedge iphase = 6 \wedge hdot >= hdmax \rightarrow dgrnzt = grnzc1 + dgrnz + 1.0$
$cont = on \wedge iphase = 6 \wedge hdot >= hdmax \wedge itgtnz = off$
$\rightarrow dgrnzt = midval(dnzb - hdmax * dhdnz, dnzmin, dnzmax)$

So the pushout gives us the part of the old specification that is compatible with the change. What is even more interesting is that, by making the difference between the axioms in the new specification and the axioms in the pushout, we get the modifications that are not consistent with the old behavior.

In some cases, it is normal to find inconsistencies (in the CR presented here for example). It is however very interesting to be able to identify precisely the differences. This can furthermore be very useful for the verification of properties: it helps determining which properties are still true in the new system, which ones are to be re-proved.

Moreover, by using this method, we can also analyze the relationships (and find inconsistencies) between different consecutive versions of a change request or between several change requests concerning the same component. Finally, the shared specifications together with the morphisms provide traceability: any element of the signature can be traced throughout the different versions using the morphisms.

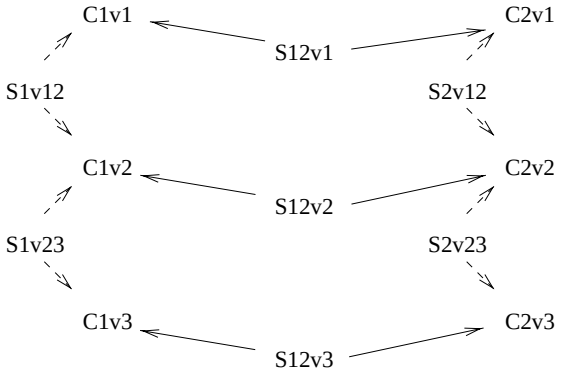
5.5 CR for the system

We now have to consider the CR at the system level.

The system is represented by several interconnected component specifications. Each component specification can be affected by the CR as explained above, but the interconnections can be affected too. It can result in new interconnections between components or in modifications of the existing interconnections.

For example, the modified version of GRALPC has new inputs and new outputs: DPSAC is a new input coming from the GTP function, GRALPC had previously no links with GTP, so we have to create a new subspecification between GRALPC and GTP to represent this.

We end up with a two dimensional framework: the system level describes the components of the system and their interconnections, the colimit can be computed to get the system specification; then for each component (and interconnecting specification) there can be several different versions of the specification (CR level) as shown on the following figure:



The two levels are independent and have different interconnection models: at the system level, attributes are local and only actions can be shared; at the CR level, everything can be shared. This results in different decompositions in axioms and co-axioms for each component depending on the level that is considered.

Some practical problems still need to be addressed. First, we have to be able to consider the two levels independently and to change the decomposition into axioms and co-axioms according to the level. A solution could be:

- to ask the user to define an interconnection model for each level (i.e. to identify the elements of the signature that can be shared by different components);
- to divide the different possible kinds of logic formulas in properties and co-properties, according to the interconnection model;
- and to consider the adequate decomposition in axioms and co-axioms for each level.

We also need to think of the practical management of the two levels: how to store and access to the different specifications and the results of the operations: colimit to get the system specification, pushout between two versions of a component.

We are planning to extend an existing tool called Moka [12]. Moka encodes general notions of category theory in ML (following [10]) and also implements temporal logic specifications, the associated morphisms and all the colimits of this category. Moka can thus handle the system level, but needs to be extended to deal with the CR level. It also needs to be adapted to the new notions of specification and morphism.

6 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a formal way of representing and analyzing changes for large systems. We have applied this approach to a change request for the Space Shuttle guidance software. We use a category theory based framework to specify the system and to represent and analyze changes. First the fact that the specification is structured and modular is important in large systems for understanding, validation and modification. Furthermore, the categorical operations are not only useful to compute the specification of the system but also to analyze changes and detect inconsistencies. Finally, we have only applied the approach to one kind of change and other applications are needed to assess the adequacy of the approach to other types of change; but the framework used here (and defined in [7]) is very general and we believe it can be adapted to different kinds of systems and interconnection models.

We are now planning to work on the implementation of such an approach. We first need to extend Moka to deal with the categorical aspects of the framework. The tool can be extended to handle the new notions of specification and morphisms. An interesting problem is to find a construction for the pushout (computation of the new co-properties). The other aspect to take into consideration at the implementation level is the practical management of the different versions of a component and of a system as mentioned in section 4. An interesting tool concerning these aspects is presented in [9].

Future work also includes verification and validation of evolving specifications. We want to combine the extensions proposed in [13] to handle properties of evolving specifications with the approach presented here. We would also like to further study how to know if a property is preserved or has to be reproved in the new version of the system; and how to reuse the proof of a property.

Finally, we are interested in the use of category theory based frameworks for inconsistency tracking [1] and formalisation of viewpoints [2].

References

1. S. Easterbrook, J. Callahan, and V. Wiels. V&V through inconsistency tracking and analysis. In *Proceedings of IWSSD 98, International Workshop on Software Specification and Design*, 1998.
2. S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering journal*, 11:31–43, 1996.
3. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985. Equations and initial semantics.
4. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990. Modules specifications and constraints.
5. J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.
6. J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 1(39), 1992.
7. A. Lopes and J. Fiadeiro. Preservation and reflection in specification. In M. Johnson, editor, *Algebraic Methodology and Software Technology 97*, volume 1349 of *LNCS*. Spinger Verlag, 1997.
8. S. Miller and K. Hoech. Specifying the mode logic of a flight guidance system in CoRE. In *Proceedings of FMSP 98, Formal Methods in Software Practice*, 1998.
9. F. Pinheiro and J. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, march 1996.
10. D. Rydeheard and R. Burstall. *Computational Category Theory*. International Series in Computer Science. Prentice Hall, 1988.
11. C. Seguin and V. Wiels. Using a Logical and Categorical Approach for the Validation of Fault-tolerant Systems. In *Proceedings of FME'96*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
12. V. Wiels. *Modularité pour la conception et la validation formelles de systèmes*. PhD thesis, ENSAE, 1997.
13. V. Wiels and S. Easterbrook. Management of evolving specifications using category theory. In *Proceedings of Automated Software Engineering'98*, 1998.
14. M. Wirsing. Algebraic specification. In *Handbook of theoretical computer science, Formal Models and Semantics*, volume B, pages 675–788. Elsevier and MIT Press, 1990.